CEWES MSRC/PET TR/98-26

The MPBench Report

by

Phillip J. Mucci
Kevin London

04h01998

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Nichols Research Corporation

The MPBench Report

Philip J. Mucci
Kevin London
mucci@cs.utk.edu
london@cs.utk.edu

March 1998

1 Introduction

MPBench is a benchmark to evaluate the performance of MPI and PVM on MPPs and clusters of workstations. It uses a flexible and portable framework to allow benchmarking of any message passing layer with similar send and receive semantics. It generates two types of reports, consisting of the raw data files and Postscript graphs. No interpretation or analysis of the data is performed, it is left entirely up to the user.

2 How it works

MPBench currently tests seven different MPI and PVM calls. MPI provides much richer functionality that PVM does, so some of the benchmarks have been implemented in terms of lower level PVM functions. The following functions are measured.

Benchmark	*Units*	*Num Procs*
Bandwidth	Megabytes/sec	2,3
Roundtrip	Transactions/sec	2,3
Application Latency	Microseconds	2,3
Broadcast	Megabytes/sec	16
Reduce	Megabytes/sec	16
AllReduce	Megabytes/sec	16

AllReduce is a reduction operation in which all tasks receive the result. This function is not available in PVM, so it is emulated by performing a *reduce* followed by a *broadcast*.

All tests are timed in the following manner.

1. Set up the test.
2. Start the timer.
3. Loop of operations over the message size as a power of two and the iteration count.
4. Verify that those operations have completed.
5. Stop the timer.
6. Compute the appropriate metric.

In MPBench, we avoid calling the timer around every operation, because this often results in the faulty reporting of data. Some of these operations on MPP's take so little time, that the accuracy and latency of accessing the system's clock would significantly affect the reported data. Thus it is only appropriate that we perform our timing operations *outside the loop*. Some MPP's and workstations have the capability to access the system's timer registers, but this is not portable and would introduce unnecessary complexity into the code to compensate for situations where the timing routines were not efficient.

For simplicity purposes, we will refer to two different types of tasks in MPBench, the master of which there is only one, and the slaves of which their may be any number. The point-to-point tests only use two tasks, a master and a slave. The other tests run with any number of slaves, the default being sixteen.

MPBench averages performance over a number of iterations. The user should be aware that MPBench will use a lower number of iterations than the one specified for certain situations. This should not effect the accuracy of the results, as the iteration count is only changed when the message lengths are prohibitively large.

By default, MPBench measures messages from 4 bytes to 16 Megabytes, in powers of two for 500 iterations. We iterate to make sure that the cache is "warmed" with the message data. This is done because applications typically communicate data soon after computing on it.

The previous version of MPBench did not pay attention to the placement of slave tasks. This caused the user to make false claims about the performance of NUMA multiprocessors like the Origin 2000 where a 2-task job will be scheduled on processors on the same physical board. MPBench now measures point-to-point performance on MPI jobs with 2 and 3 tasks.

For the 3-task case, task 1 (of 0, 1 and 2) remains idle until the end of the run. In fact, this arrangement does not guarantee that the remote processor is offboard, as that is dictated by the MPI environment. However, our measurements seem to indicate that task 2 in a 3-task job always performs slightly slower than task 1 of a 2-task job. We attribute this to extra time required to arbitrate and negotiate the link.

2.1 Notes on PVM

PVM has a number of options to accelerate performance. For this benchmark, we are interested in the optimal performance of the machine. In order to do so, we use the following options where possible:

- *Direct Routing* - This option sets up direct TCP/IP connections between processes as opposed to forward messages via daemon processes running on each node. It is established by calling `pvm_setopt(PvmRoute, PvmRouteDirect)` .
- *In-place Packing* - All messages in PVM must be *packed* into PVM buffers before transmission. This option tells PVM not to perform any additional copies of the data, but to transmit the data directly out of the application's buffer. This option precludes the use of any data-translation with PVM. By default, in-place packing is used whenever `pvm_psend()` is called, which is what is used in this program.

2.2 Notes on MPI

There are many different send and receive calls in MPI each with different semantics for usage and completion. Here we focus on the *default* mode of sending. This means we are not using any *nonblocking* or *immediate* communication calls. Each MPI implementation handles the default mode a bit differently, but the algorithm is usually a derivative of the following.

```
send first chunk of message
if message is larger than size N
  wait for reply and destination address Y
    send rest of message directly to address Y
else
  if more to send
    send rest of message
endif
```

MPI does this to avoid unnecessary copies of the data, which usually dominates the cost of any communication layer. The receiving process will buffer a limited amount of data before informing the sender of the destination address in the application. This way, a large message is received directly into the application's data structure rather than being held in temporary storage like with PVM. The problem with this is that for large messages, `sends` cannot complete before their corresponding `receives`. This introduces possibly synchronization and portability problems.

2.3 Bandwidth

MPBench measures bandwidth with a doubly-nested loop. The outer loop varies the message size, and the inner loop measures the send operation over the iteration count. After the iteration count is reached, the slave process *acknowledges* the data it has received by sending a four byte message back to the master. This informs the sender when the slaves have completely finished receiving their data and are ready to proceed. This is necessary, because the send on the master may complete before the matching receive does on the slave. This exchange does introduce additional overhead, but given a large iteration count, its effect is minimal.

The master's pseudocode for this test is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    send(message size)
  recv(4)
  stop timer
```

The slave's pseudocode is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    recv(message size)
  send(4)
  stop timer
```

2.4 Roundtrip

Roundtrip times are measured in much the same way as bandwidth, except that, the slave process, after receiving the message, echoes it back to the master. This benchmark is often referred to as *ping-pong*. Here our metric is transactions per second, which is a common metric for database and server applications. No acknowledgment is needed with this test as it is implicit given its semantics.

The master's pseudocode for this test is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    send(message size)
    recv(message size)
  stop timer
```

The slave's pseudocode is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    recv(message size)
    send(message size)
  stop timer
```

2.5 Application Latency

Application latency is something relatively unique to MPBench. This benchmark can properly be described as one that measures the time for an application to issue a `send` and continue computing. The results for this test vary greatly given how the message passing layer is implemented. For example, PVM will buffer all messages for transmission, regardless of whether or not the remote node is ready to receive the data. MPI on the other hand, will not buffer messages over a certain size, and thus will block until the remote process has executed some form of a `receive`. This benchmark is the same as bandwidth except that we do not acknowledge the data and we report our results in units of time. This benchmark very much represents the time our application will be waiting to do useful work while communicating.

The master's pseudocode for this test is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    send(message size)
  stop timer
```

The slave's pseudocode is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    recv(message size)
  stop timer
```

2.6 Broadcast and Reduce

The two functions are very heavily used in many parallel applications. Essentially these operations are mirror images of one another, the difference being that reduce reverses the direction of communication and performs some computation with the data during intermediate steps. Both of these benchmarks return the number of megabytes per second computed from the iteration count and the length argument given to function call.

With PVM, broadcast and reduce will not complete unless the application has performed a barrier immediately prior to the operation. Thus, with PVM both of these tests include the cost of a barrier operation every iteration.

Here is the pseudocode for both the master and the slave:

```
do over all message sizes
  start timer
  do over iteration count
    reduce or broadcast(message size)
  stop timer
```

2.7 AllReduce

AllReduce is a derivative of an all-to-all communication, where every process has data for every other. While this operation could easily be implemented with a reduce followed by a broadcast, that would be highly inefficient for large message sizes. The PVM version of this test does this exactly, plus an additional barrier call. The goal of including this benchmark

is to spot poor implementations so that the application engineer might be able to restructure his communication.

Here is the pseudocode for both the master and the slave:

```
do over all message sizes
  start timer
  do over iteration count
    allreduce(message size)
  stop timer
```

3 Using MPBench

3.1 Obtain the Distribution

Download the latest release from either of the following URLs:

`http://www.cs.utk.edu/~mucci/mpbench`
`ftp://cs.utk.edu/pub/mucci/mpbench.tar.gz`

Now unpack the installation using `gzip` and `tar`.

```
pebbles> gzip -dc mpbench.tar.gz | tar xvf -
pebbles> cd mpbench
pebbles>  ls
CVS/            README          index.html*     make.def        mpbench.c
Makefile        conf/           lib/            make_graphs.sh* samples/
```

3.2 Build the distribution

First we must configure the build for our machine, operating system and release of MPI. All configurations support PVM. Before configuration, `make` with no arguments lists the possible targets.

```
pebbles> make
Please configure using one of the following targets:

sp2
t3e
```

```
pca-r8k
o2k
sgi32-lam
sgi64-lam
linux-lam
linux-mpich
solaris-mpich
sun4-mpich
alpha
```

Configure the build. Here, we are using a SunOS workstation, using MPICH as our MPI implementation.

```
pebbles> make sun4-mpich
rm -f make.def
ln -s conf/make.def.sun4-mpich make.def
```

Now look at the available targets, and build one.

```
pebbles:mpbench> make
Please use one of the following targets:

mpi,pvm,all
run-mpi,run-pvm,run
graph-mpi,graph-pvm,graphs

pebbles> make all
gcc   -O2 -DINLINE -I/src/icl/MPI/mpich/include -DMPI -c mpbench.c -o mpibench.o
gcc   -O2 -DINLINE ./mpibench.o -o mpi_bench -L/src/icl/MPI/mpich/lib/sun4/ch_p4 -lmpi
sed -e "s:MPIRUNCMD:mpirun:g" < lib/mpibench.sh | \
sed -e "s:MPIRUNOPTS::g" | \
sed -e "s:MPIRUNPROCS:-np:g" | \
sed -e "s:MPIHOSTFILE:-machinefile:g" | \
sed -e "s:MPIRUNPOSTOPTS:mpi_bench:g" > mpi_bench.sh
chmod +x mpi_bench.sh
gcc   -I/shag/homes/mucci/pvm3/include  -O2 -DINLINE -DPVM -c mpbench.c -o pvmbench.o
gcc   -O2 -DINLINE ./pvmbench.o -o /shag/homes/mucci/pvm3/bin/SUN4/pvm_bench \\
\ -L/shag/homes/mucci/pvm3/lib/SUN4 -lpvm3 -lgpvm3
```

```
cp lib/pvmbench.sh /shag/homes/mucci/pvm3/bin/SUN4/pvm_bench.sh
chmod +x /shag/homes/mucci/pvm3/bin/SUN4/pvm_bench.sh
```

3.3 Running MPBench

While MPBench can be run from the command line, it is designed to be run from the `Makefile`. When running MPI, sometimes it is required that you set up a hostfile containing the names of the hosts on which to run the processes. If your installation requires a hostfile, MPBench will tell you. If that happens, please check your `mpirun` man page for the format. The resulting datafiles for each of the runs will be left in `mpbench/results/<OS>-<HOSTNAME>_<API>_<test`

```
pebbles> make run-mpi
Testing mpirun...
Current value is: mpirun  -machinefile \\
\ /shag/homes/mucci/mpibench-hostfile -np 2 mpi_bench
%Measuring barrier for 500 iterations with 2 tasks...
%Measuring barrier for 500 iterations with 4 tasks...
%Measuring barrier for 500 iterations with 8 tasks...
%Measuring barrier for 500 iterations with 16 tasks...
Measuring latency for 500 iterations...
Measuring roundtrip for 500 iterations...
Measuring bandwidth for 500 iterations...
Measuring broadcast for 500 iterations with 16 tasks...
Measuring reduce for 500 iterations with 16 tasks...
Measuring allreduce for 500 iterations with 16 tasks...
```

Now we plot the results with GNUplot. If GNUplot is not available on your system, perform the following.

- Unpack the distribution on a machine that does.
- Copy your results files to the new machine in the MPBench directory.
- Execute make_graphs.sh with the common prefix of your datafiles.

Normally, we can just do one of the following:

```
make graphs
make graph-mpi
make graph-pvm

pebbles> make graph-mpi
```

The graphs will be left in the `results` directory.

4 Results on the CEWES MSRC Machines

The following graphs are taken from our runs on each of the CEWES MSRC machines during dedicated time. Those machines are the SGI Origin 2000, the IBM SP and the Cray T3E.

Machine	*Cache*
SGI Origin 2000	32K,4MB
IBM SP	128K
Cray T3E	8K,96K

4.1 Latency

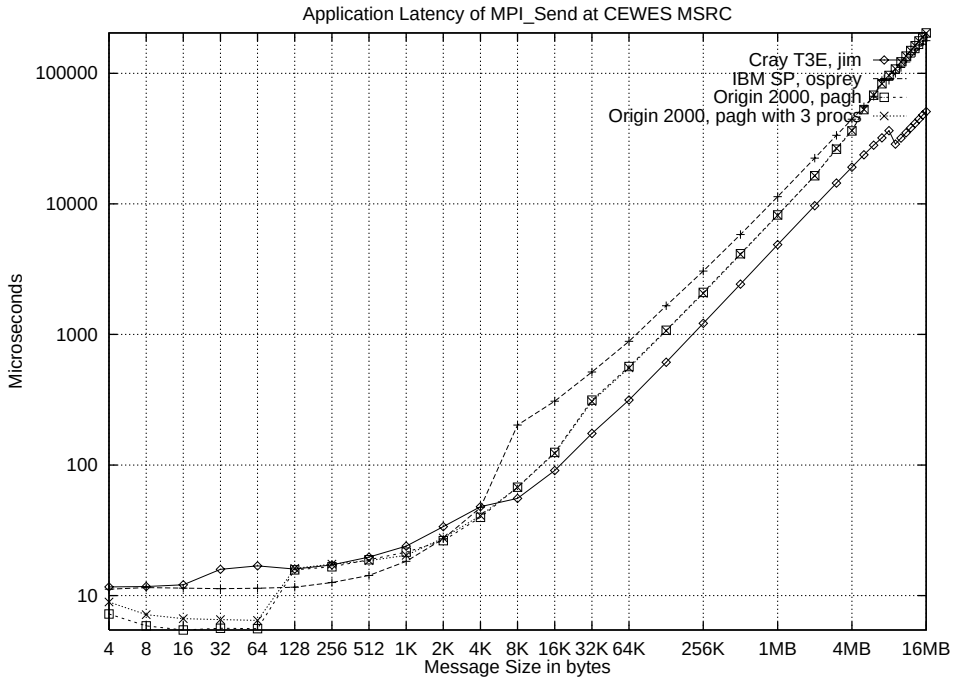

Figure 1: Application Latency of Send

In the Figure 1 we see three interesting performance variations. First note the jump in latency of the O2K when the message is greater than 64 bytes. This is likely due to space allocated in the header exchanged between two processes. Many message passing systems allocate space in the header for a small payload so only one exchange is required. Next we note the jump in latency on the SP for messages larger than 4096 bytes. This is the point where IBM's MPI switches to a rendezvous protocol. This is tunable from the command line for `poe`, IBM's version of `mpirun`, with the `-eagerlimit` argument. It is also tunable with the `MP_EAGERLIMIT` environment variable. We recommend setting this to 16384 bytes for all runs. In fact, IBM does this when running parallel benchmarks. Also, note the falloff in performance at 8MB on the T3E. This is found throughout all our communication graphs and we are currently unable to explain it. On the Origin we see a steady increase in the latency corresponding to the message size. The Origin exhibits extremely low latencies until they exceed a cache size. These latencies suggest that the Origin might be suitable for applications that do a lot of data exchange in small quantities. When the message exceeds the 64 byte cache line on the Origin, it appears that some expensive routine is being called increasing

the latency more than twenty-fold. As far as the differences between Origin processors on and off the node, it appears that the only time this is a factor is when messages are smaller than the cache line size. Otherwise, the transmission time appears to be dominated by the memory controller not the communication link.

4.2 Roundtrip

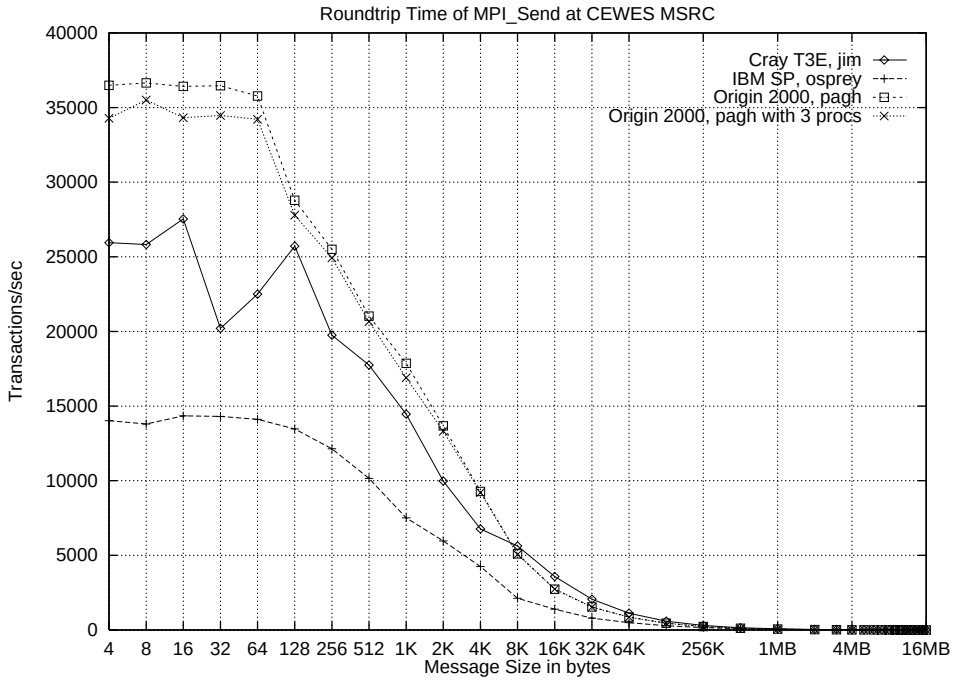

Figure 2: Roundtrip Time of Ping-Pong

Figure 2 is a graph of the average roundtrip time for a message exchange. This test is also commonly referred to as ping-pong. For small messages, roundtrip time is largely dominated by protocol overheads and the method of access to the network hardware. Notice in Figure 3 that while the bandwidth for the T3E are higher than for the Origin, the Origin still outperforms it. An inversion takes place at 8K messages between the Origin and the T3E. We deduce that the Origin with its distributed shared memory hardware provides a very lightweight method of accessing remote memory. 8K is the page size of the Origin 2000, so it is not surprising that a penalty is paid after crossing that boundary. For the 3-task test case it appears that only very small message sizes are affected. At larger messages, the raw link speed of the T3E clearly dominates, while the performance of the SP and the Origin falters.

4.3 Bandwidth

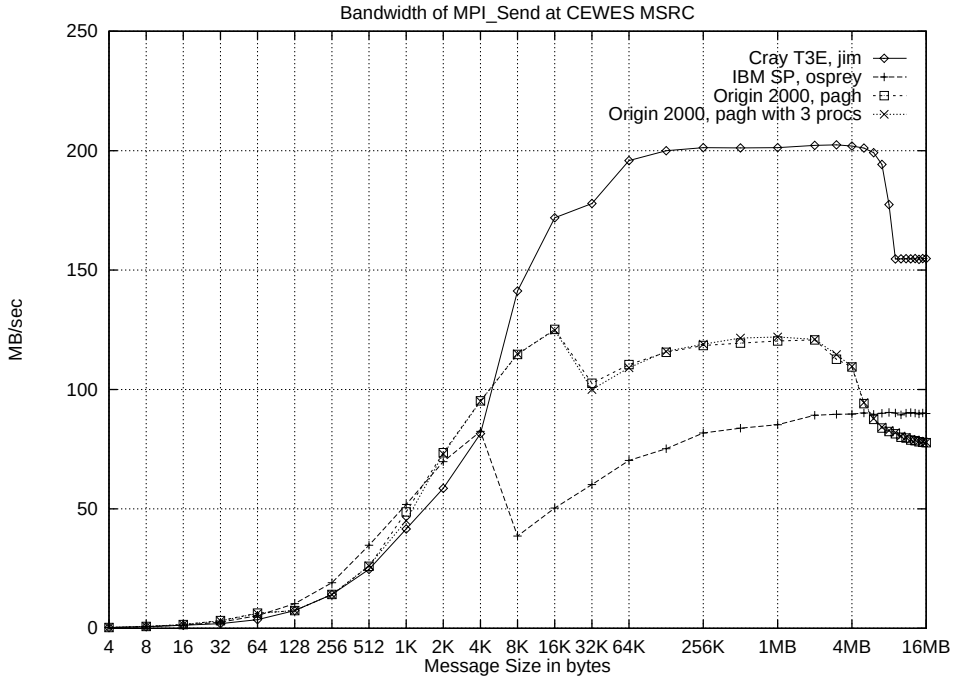

Figure 3: Bandwidth of Send

In Figure 3, we note the dramatic effect of MPI's rendezvous protocol on all three machines. The tradeoff is latency for bandwidth, but it doesn't always appear to be valid. As mentioned, the SP has a rather small limit of 4K, thus responsible for the fifty percent falloff at larger message sizes. The Origin and the T3E both have an eager limit set to 16K, with only the Origin suffering a loss in performance. Also of interest is the effect that caching has on the Origin. As mentioned, these tests are repeated a number of times, so most of the data will lie in the Origin's large 4MB level two cache. Note that for larger sizes, its performance suffers severely. We recommend that users raise the default eager limit to 32K for their runs.

4.4 Broadcast

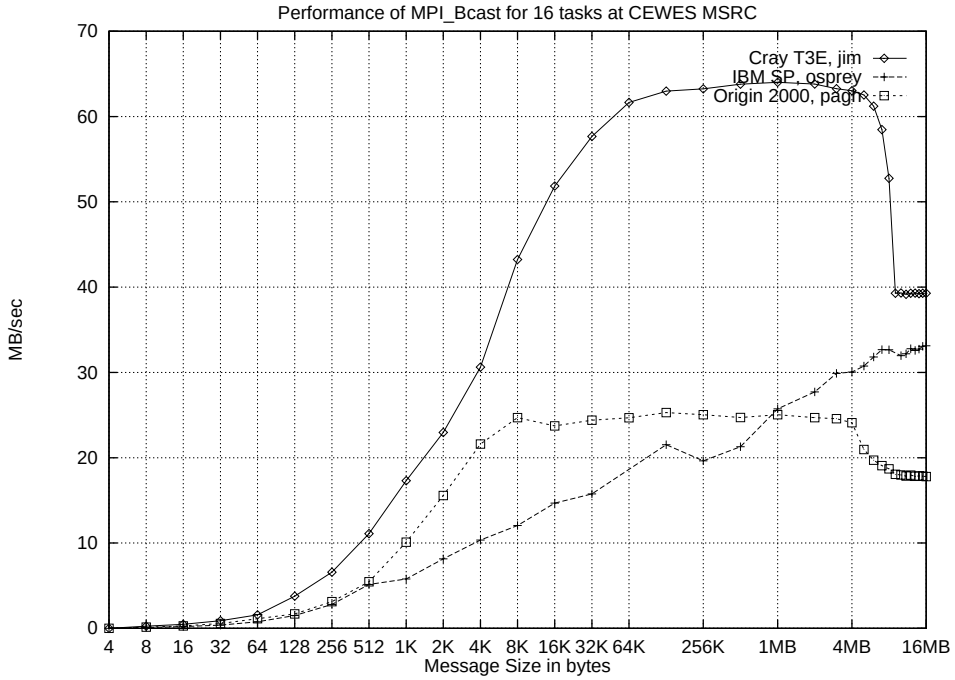

Figure 4: Broadcast Performance

For Figure 4, we again note the dramatic drop-off found at the 8MB message size on the T3E. For the Origin, we also notice the effect of cache. The user should be aware that this test also includes the time for every task to send an acknowledge back to the master. In Figure 5 we show the time steps for this test on an eight processor system. If we assume a left-to-right ordered binary tree distribution algorithm, we receive our first acknowledgement after $\log_2(8) - 1$ or 2 full sends. Our last acknowledgement arrives on the rightmost branch after $\log_2(8) + 1$ or 4 full sends have completed. Note that in this test, the timer does not stop until we receive the last acknowledgment. Relating our broadcast performance to our bandwidth requires understanding that we must execute at least one send to transmit the data. Thus to compare broadcast to bandwidth performance we must take into account the first send. Therefore broadcast performance should be $1/\log_2(p)$ that of bandwidth. This only holds true for machines that have no hardware assisted broadcast. Our results seem to agree with this model.

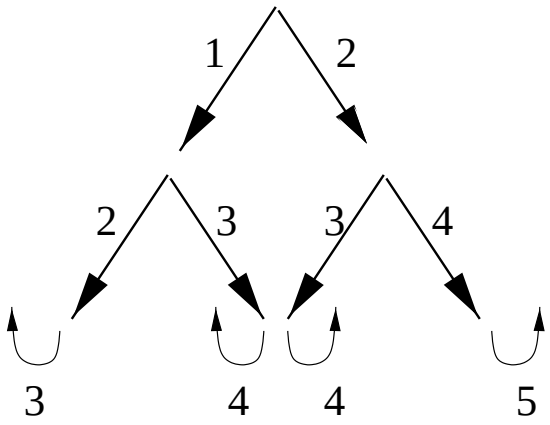

Figure 5: Message time steps tree for 8 nodes

4.5 Reduce

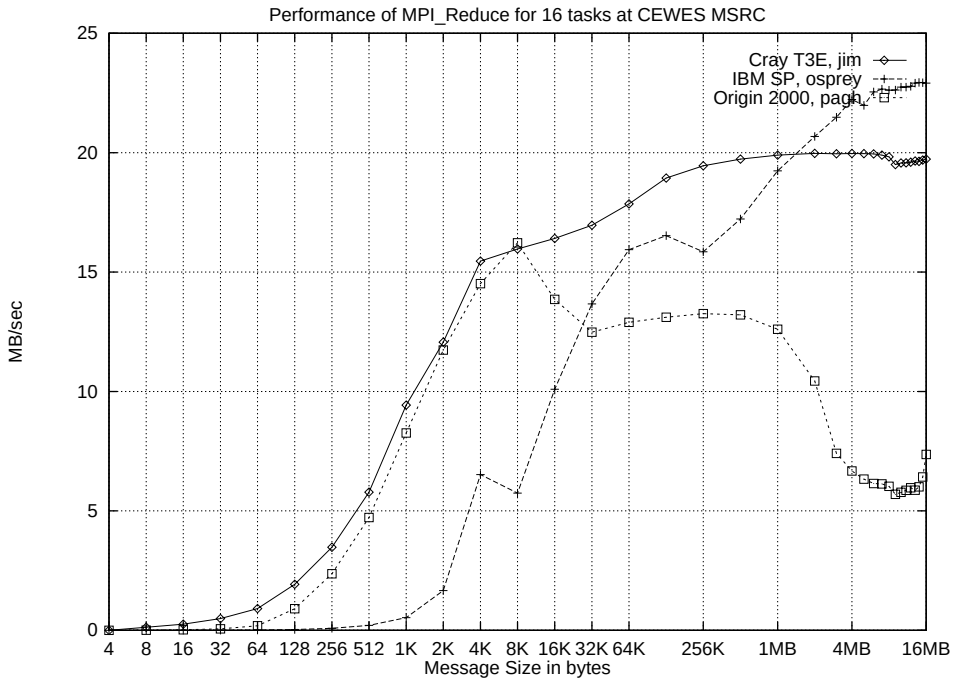

Figure 6: Reduce Performance

In Figure 6 we plot the performance of a reduction operation. Due to a change in the MPI protocol, we would expect a graduate increase in performance after the initial drop. However, because this is an iterated test, the cache may be hiding the effects of the MPI protocol and exaggerating the cost of a distributed page fault. The large hump is where the message fits into the level two cache. For the SP, the dip at the 4K message size is again related to the small eager limit. Performance of the T3E increases steadily and levels off around 20MB/sec. Notice the lack of a significant falloff at larger messages on the T3E. Also notice how poorly the T3E performs in relation to Figure 4.

4.6 AllReduce

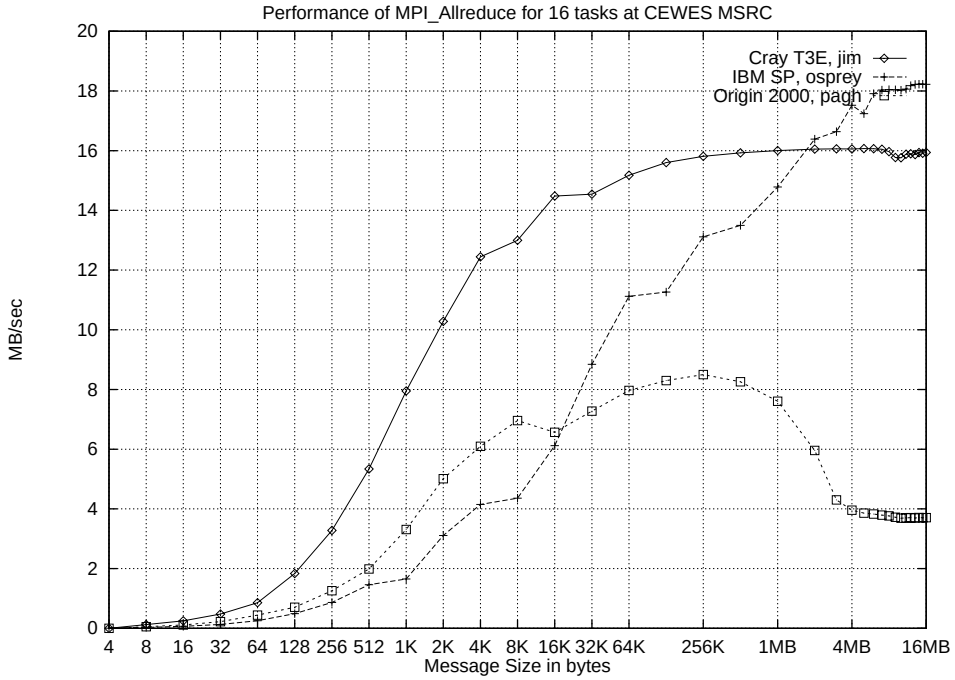

Figure 7: AllReduce Performance

For Figure 7, we again notice the dramatic effect caching has on the Origin with performance falling off around the 8MB mark. Comparing this graph with that of Figure 6, we note that the SP and the T3E perform about twenty percent worse on Allreduce than on Reduce. The Origin performs more than thirty percent worse, which is perhaps an architectural problem related to network contention.

5 Future work

- Provide option for adjusting the test space.
- Provide the option for measuring a specific message size.
- Provide an option for cache flushing between transmission.
- Use specialized, high-resolution timers where available.
- Add benchmarks for `MPI_Isend`, `MPI_Irecv`, `MPI_Irsend` and `MPI_Alltoall`.
- Standardize configuration with GNU *autoconf*.
- Grab machine configuration and store it with each run.
- Standardize data/graph naming scheme with timestamp.

6 References

PVM - Parallel Virtual Machine Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, MIT Press, Cambridge, 1994

MPI - The Complete Reference Marc Snir, Steve W. Otto, Steve Huss-Lederman, David W. Walker, Jack Dongarra, MIT Press, Cambridge, 1996